# Node.js v9.2.0 Documentation



## Table of Contents

- util.isPrimitive(object) deprecated
- util.isRegExp(object) deprecated
- util.isString(object) deprecated
- util.isSymbol(object) deprecated
- util.isUndefined(object) deprecated
- util.log(string) deprecated
- util.print([...strings]) deprecated
- util.puts([...strings]) deprecated

# Util #

Stability: 2 - Stable

The `util` module is primarily designed to support the needs of Node.js' own internal APIs. However, many of the utilities are useful for application and module developers as well. It can be accessed using:

```
const util = require('util');
```

# util.callbackify(original) #

Added in: v8.2.0

- `original` `<Function>` An `async` function
- Returns: `<Function>` a callback style function

Takes an `async` function (or a function that returns a Promise) and returns a function following the Node.js error first callback style. In the callback, the first argument will be the rejection reason (or `null` if the Promise resolved), and the second argument will be the resolved value.

For example:

```
const util = require('util');

async function fn() {
  return await Promise.resolve('hello world');
}
const callbackFunction = util.callbackify(fn);

callbackFunction((err, ret) => {
  if (err) throw err;
  console.log(ret);
});
```

Will print:

```
hello world
```

*Note*:

- The callback is executed asynchronously, and will have a limited stack trace. If the callback throws, the process will emit an `'uncaughtException'` event, and if not handled will exit.
- Since `null` has a special meaning as the first argument to a callback, if a wrapped function rejects a `Promise` with a falsy value as a reason, the value is wrapped in an `Error` with the original value stored in a field named `reason`.

```
function fn() {
  return Promise.reject(null);
}
const callbackFunction = util.callbackify(fn);

callbackFunction((err, ret) => {
  // When the Promise was rejected with `null` it is wrapped with an Error and
  // the original value is stored in `reason`.
  err && err.hasOwnProperty('reason') && err.reason === null;  // true
});
```

# util.debuglog(section) #

Added in: v0.11.3

- `section` `<string>` A string identifying the portion of the application for which the `debuglog` function is being created.
- Returns: `<Function>` The logging function

The `util.debuglog()` method is used to create a function that conditionally writes debug messages to `stderr` based on the existence of the `NODE_DEBUG` environment variable. If the `section` name appears within the value of that environment variable, then the returned function operates similar to `console.error()`. If not, then the returned function is a no-op.

For example:

```
const util = require('util');
const debuglog = util.debuglog('foo');

debuglog('hello from foo [%d]', 123);
```

If this program is run with `NODE_DEBUG=foo` in the environment, then it will output something like:

```
FOO 3245: hello from foo [123]
```

where `3245` is the process id. If it is not run with that environment variable set, then it will not print anything.

Multiple comma-separated `section` names may be specified in the `NODE_DEBUG` environment variable. For example: `NODE_DEBUG=fs,net,tls`.

# util.deprecate(function, string) #

Added in: v0.8.0

The `util.deprecate()` method wraps the given `function` or class in such a way that it is marked as deprecated.

```
const util = require('util');

exports.puts = util.deprecate(function() {
  for (let i = 0, len = arguments.length; i < len; ++i) {
    process.stdout.write(arguments[i] + '\n');
  }
}, 'util.puts: Use console.log instead');
```

When called, `util.deprecate()` will return a function that will emit a `DeprecationWarning` using the `process.on('warning')` event. By default, this warning will be emitted and printed to `stderr` exactly once, the first time it is called. After the warning is emitted, the wrapped `function` is called.

If either the `--no-deprecation` or `--no-warnings` command line flags are used, or if the `process.noDeprecation` property is set to `true` *prior* to the first deprecation warning, the `util.deprecate()` method does nothing.

If the `--trace-deprecation` or `--trace-warnings` command line flags are set, or the `process.traceDeprecation` property is set to `true`, a warning and a stack trace are printed to `stderr` the first time the deprecated function is called.

If the `--throw-deprecation` command line flag is set, or the `process.throwDeprecation` property is set to `true`, then an exception will be thrown when the deprecated function is called.

The `--throw-deprecation` command line flag and `process.throwDeprecation` property take precedence over `--trace-deprecation` and `process.traceDeprecation`.

# util.format(format[, ...args]) #

▶ History

- `format` `<string>` A `printf`-like format string.

The `util.format()` method returns a formatted string using the first argument as a `printf`-like format.

The first argument is a string containing zero or more *placeholder* tokens. Each placeholder token is replaced with the converted value from the corresponding argument. Supported placeholders are:

- `%s` - String.
- `%d` - Number (integer or floating point value).
- `%i` - Integer.
- `%f` - Floating point value.
- `%j` - JSON. Replaced with the string `'[Circular]'` if the argument contains circular references.
- `%o` - Object. A string representation of an object with generic JavaScript object formatting. Similar to `util.inspect()` with options `{ showHidden: true, depth: 4, showProxy: true }`. This will show the full object including non-enumerable symbols and properties.

- `%O` - Object. A string representation of an object with generic JavaScript object formatting. Similar to `util.inspect()` without options. This will show the full object not including non-enumerable symbols and properties.
- `%%` - single percent sign ( `'%'` ). This does not consume an argument.

If the placeholder does not have a corresponding argument, the placeholder is not replaced.

```
util.format('%s:%s', 'foo');
// Returns: 'foo:%s'
```

If there are more arguments passed to the `util.format()` method than the number of placeholders, the extra arguments are coerced into strings then concatenated to the returned string, each delimited by a space. Excessive arguments whose `typeof` is `'object'` or `'symbol'` (except `null` ) will be transformed by `util.inspect()` .

```
util.format('%s:%s', 'foo', 'bar', 'baz'); // 'foo:bar baz'
```

If the first argument is not a string then `util.format()` returns a string that is the concatenation of all arguments separated by spaces. Each argument is converted to a string using `util.inspect()` .

```
util.format(1, 2, 3); // '1 2 3'
```

If only one argument is passed to `util.format()` , it is returned as it is without any formatting.

```
util.format('%% %s'); // '%% %s'
```

# util.inherits(constructor, superConstructor) #

▶ History

*Note*: Usage of `util.inherits()` is discouraged. Please use the ES6 `class` and `extends` keywords to get language level inheritance support. Also note that the two styles are semantically incompatible.

- `constructor <Function>`
- `superConstructor <Function>`

Inherit the prototype methods from one constructor into another. The prototype of `constructor` will be set to a new object created from `superConstructor`.

As an additional convenience, `superConstructor` will be accessible through the `constructor.super_` property.

```
const util = require('util');
const EventEmitter = require('events');

function MyStream() {
  EventEmitter.call(this);
}

util.inherits(MyStream, EventEmitter);

MyStream.prototype.write = function(data) {
  this.emit('data', data);
};

const stream = new MyStream();

console.log(stream instanceof EventEmitter); // true
```

```
console.log(MyStream.super_ === EventEmitter); // true

stream.on('data', (data) => {
  console.log(`Received data: "${data}"`);
});
stream.write('It works!'); // Received data: "It works!"
```

ES6 example using `class` and `extends`

```
const EventEmitter = require('events');

class MyStream extends EventEmitter {
  write(data) {
    this.emit('data', data);
  }
}

const stream = new MyStream();

stream.on('data', (data) => {
  console.log(`Received data: "${data}"`);
});
stream.write('With ES6');
```

# util.inspect(object[, options]) #

▶ History

- `object` <any> Any JavaScript primitive or Object.
- `options` <Object>
  - `showHidden` <boolean> If `true`, the `object`'s non-enumerable symbols and properties will be included in the formatted result. Defaults to `false`.
  - `depth` <number> Specifies the number of times to recurse while formatting the `object`. This is useful for inspecting large complicated objects. Defaults to `2`. To make it recurse indefinitely pass `null`.
  - `colors` <boolean> If `true`, the output will be styled with ANSI color codes. Defaults to `false`. Colors are customizable, see Customizing `util.inspect` colors.
  - `customInspect` <boolean> If `false`, then custom `inspect(depth, opts)` functions exported on the `object` being inspected will not be called. Defaults to `true`.
  - `showProxy` <boolean> If `true`, then objects and functions that are `Proxy` objects will be introspected to show their `target` and `handler` objects. Defaults to `false`.
  - `maxArrayLength` <number> Specifies the maximum number of array and `TypedArray` elements to include when formatting. Defaults to `100`. Set to `null` to show all array elements. Set to `0` or negative to show no array elements.
  - `breakLength` <number> The length at which an object's keys are split across multiple lines. Set to `Infinity` to format an object as a single line. Defaults to 60 for legacy compatibility.

The `util.inspect()` method returns a string representation of `object` that is primarily useful for debugging. Additional `options` may be passed that alter certain aspects of the formatted string.

The following example inspects all properties of the `util` object:

```
const util = require('util');

console.log(util.inspect(util, { showHidden: true, depth: null }));
```

Values may supply their own custom `inspect(depth, opts)` functions, when called these receive the current `depth` in the recursive inspection, as well as the options object passed to `util.inspect()`.

## Customizing `util.inspect` colors #

Color output (if enabled) of `util.inspect` is customizable globally via the `util.inspect.styles` and `util.inspect.colors` properties.

`util.inspect.styles` is a map associating a style name to a color from `util.inspect.colors`.

The default styles and associated colors are:

- `number` - `yellow`
- `boolean` - `yellow`
- `string` - `green`
- `date` - `magenta`
- `regexp` - `red`
- `null` - `bold`
- `undefined` - `grey`
- `special` - `cyan` (only applied to functions at this time)
- `name` - (no styling)

The predefined color codes are: `white`, `grey`, `black`, `blue`, `cyan`, `green`, `magenta`, `red` and `yellow`. There are also `bold`, `italic`, `underline` and `inverse` codes.

Color styling uses ANSI control codes that may not be supported on all terminals.

## Custom inspection functions on Objects #

Objects may also define their own `[util.inspect.custom](depth, opts)` (or the equivalent but deprecated `inspect(depth, opts)`) function that `util.inspect()` will invoke and use the result of when inspecting the object:

```
const util = require('util');

class Box {
  constructor(value) {
    this.value = value;
  }

  [util.inspect.custom](depth, options) {
    if (depth < 0) {
      return options.stylize('[Box]', 'special');
    }

    const newOptions = Object.assign({}, options, {
      depth: options.depth === null ? null : options.depth - 1
    });

    // Five space padding because that's the size of "Box< ".
    const padding = ' '.repeat(5);
    const inner = util.inspect(this.value, newOptions)
                      .replace(/\n/g, `\n${padding}`);
    return `${options.stylize('Box', 'special')}< ${inner} >`;
  }
}

const box = new Box(true);
```

```
util.inspect(box);
// Returns: "Box< true >"
```

Custom `[util.inspect.custom](depth, opts)` functions typically return a string but may return a value of any type that will be formatted accordingly by `util.inspect()`.

```
const util = require('util');

const obj = { foo: 'this will not show up in the inspect() output' };
obj[util.inspect.custom] = function(depth) {
  return { bar: 'baz' };
};

util.inspect(obj);
// Returns: "{ bar: 'baz' }"
```

## util.inspect.custom #

Added in: v6.6.0

A Symbol that can be used to declare custom inspect functions, see Custom inspection functions on Objects.

## util.inspect.defaultOptions #

Added in: v6.4.0

The `defaultOptions` value allows customization of the default options used by `util.inspect`. This is useful for functions like `console.log` or `util.format` which implicitly call into `util.inspect`. It shall be set to an object containing one or more valid `util.inspect()` options. Setting

option properties directly is also supported.

```
const util = require('util');
const arr = Array(101).fill(0);

console.log(arr); // logs the truncated array
util.inspect.defaultOptions.maxArrayLength = null;
console.log(arr); // logs the full array
```

# util.isDeepStrictEqual(val1, val2) #

Added in: v9.0.0

- `val1` <any>
- `val2` <any>
- Returns: <boolean>

Returns `true` if there is deep strict equality between `val` and `val2`. Otherwise, returns `false`.

See `assert.deepStrictEqual()` for more information about deep strict equality.

# util.promisify(original) #

Added in: v8.0.0

- `original` <Function>
- Returns: <Function>

Takes a function following the common Node.js callback style, i.e. taking a `(err, value) => ...` callback as the last argument, and returns a version that returns promises.

For example:

```
const util = require('util');
const fs = require('fs');

const stat = util.promisify(fs.stat);
stat('.').then((stats) => {
  // Do something with `stats`
}).catch((error) => {
  // Handle the error.
});
```

Or, equivalently using `async function` s:

```
const util = require('util');
const fs = require('fs');

const stat = util.promisify(fs.stat);

async function callStat() {
  const stats = await stat('.');
  console.log(`This directory is owned by ${stats.uid}`);
}
```

If there is an `original[util.promisify.custom]` property present, `promisify` will return its value, see Custom promisified functions .

`promisify()` assumes that `original` is a function taking a callback as its final argument in all cases, and the returned function will result in undefined behavior if it does not.

## Custom promisified functions #

Using the `util.promisify.custom` symbol one can override the return value of `util.promisify()`:

```
const util = require('util');

function doSomething(foo, callback) {
  // ...
}

doSomething[util.promisify.custom] = function(foo) {
  return getPromiseSomehow();
};

const promisified = util.promisify(doSomething);
console.log(promisified === doSomething[util.promisify.custom]);
// prints 'true'
```

This can be useful for cases where the original function does not follow the standard format of taking an error-first callback as the last argument.

For example, with a function that takes in `(foo, onSuccessCallback, onErrorCallback)`:

```
doSomething[util.promisify.custom] = function(foo) {
  return new Promise(function(resolve, reject) {
```

```
    doSomething(foo, resolve, reject);
  });
};
```

## util.promisify.custom #

Added in: v8.0.0

- `<symbol>`

A Symbol that can be used to declare custom promisified variants of functions, see Custom promisified functions .

## Class: util.TextDecoder #

Added in: v8.3.0

An implementation of the WHATWG Encoding Standard `TextDecoder` API.

```
const decoder = new TextDecoder('shift_jis');
let string = '';
let buffer;
while (buffer = getNextChunkSomehow()) {
  string += decoder.decode(buffer, { stream: true });
}
string += decoder.decode(); // end-of-stream
```

## WHATWG Supported Encodings #

Per the WHATWG Encoding Standard , the encodings supported by the `TextDecoder` API are outlined in the tables below. For each encoding, one or more aliases may be used.

Different Node.js build configurations support different sets of encodings. While a very basic set of encodings is supported even on Node.js builds without ICU enabled, support for some encodings is provided only when Node.js is built with ICU and using the full ICU data (see Internationalization ).

## Encodings Supported Without ICU #

| Encoding | Aliases |
| --- | --- |
| `'utf-8'` | `'unicode-1-1-utf-8'`, `'utf8'` |
| `'utf-16le'` | `'utf-16'` |

## Encodings Supported by Default (With ICU) #

| Encoding | Aliases |
| --- | --- |
| `'utf-8'` | `'unicode-1-1-utf-8'`, `'utf8'` |
| `'utf-16le'` | `'utf-16'` |
| `'utf-16be'` | |

## Encodings Requiring Full ICU Data #

| Encoding | Aliases |
| --- | --- |
| 'ibm866' | '866', 'cp866', 'csibm866' |
| 'iso-8859-2' | 'csisolatin2', 'iso-ir-101', 'iso8859-2', 'iso88592', 'iso_8859-2', 'iso_8859-2:1987', 'l2', 'latin2' |
| 'iso-8859-3' | 'csisolatin3', 'iso-ir-109', 'iso8859-3', 'iso88593', 'iso_8859-3', 'iso_8859-3:1988', 'l3', 'latin3' |
| 'iso-8859-4' | 'csisolatin4', 'iso-ir-110', 'iso8859-4', 'iso88594', 'iso_8859-4', 'iso_8859-4:1988', 'l4', 'latin4' |
| 'iso-8859-5' | 'csisolatincyrillic', 'cyrillic', 'iso-ir-144', 'iso8859-5', 'iso88595', 'iso_8859-5', 'iso_8859-5:1988' |
| 'iso-8859-6' | 'arabic', 'asmo-708', 'csiso88596e', 'csiso88596i', 'csisolatinarabic', 'ecma-114', 'iso-8859-6-e', 'iso-8859-6-i', 'iso-ir-127', 'iso8859-6', 'iso88596', 'iso_8859-6', 'iso_8859-6:1987' |
| 'iso-8859-7' | 'csisolatingreek', 'ecma-118', 'elot_928', 'greek', 'greek8', 'iso-ir-126', 'iso8859-7', 'iso88597', 'iso_8859-7', 'iso_8859-7:1987', 'sun_eu_greek' |
| 'iso-8859-8' | 'csiso88598e', 'csisolatinhebrew', 'hebrew', 'iso-8859-8-e', 'iso-ir-138', 'iso8859-8', 'iso88598', 'iso_8859-8', 'iso_8859-8:1988', 'visual' |
| 'iso-8859-8-i' | 'csiso88598i', 'logical' |

| Encoding | Aliases |
| --- | --- |
| 'iso-8859-10' | 'csisolatin6', 'iso-ir-157', 'iso8859-10', 'iso885910', 'l6', 'latin6' |
| 'iso-8859-13' | 'iso8859-13', 'iso885913' |
| 'iso-8859-14' | 'iso8859-14', 'iso885914' |
| 'iso-8859-15' | 'csisolatin9', 'iso8859-15', 'iso885915', 'iso_8859-15', 'l9' |
| 'koi8-r' | 'cskoi8r', 'koi', 'koi8', 'koi8_r' |
| 'koi8-u' | 'koi8-ru' |
| 'macintosh' | 'csmacintosh', 'mac', 'x-mac-roman' |
| 'windows-874' | 'dos-874', 'iso-8859-11', 'iso8859-11', 'iso885911', 'tis-620' |
| 'windows-1250' | 'cp1250', 'x-cp1250' |
| 'windows-1251' | 'cp1251', 'x-cp1251' |

| Encoding | Aliases |
| --- | --- |
| 'windows-1252' | 'ansi_x3.4-1968', 'ascii', 'cp1252', 'cp819', 'csisolatin1', 'ibm819', 'iso-8859-1', 'iso-ir-100', 'iso8859-1', 'iso88591', 'iso_8859-1', 'iso_8859-1:1987', 'l1', 'latin1', 'us-ascii', 'x-cp1252' |
| 'windows-1253' | 'cp1253', 'x-cp1253' |
| 'windows-1254' | 'cp1254', 'csisolatin5', 'iso-8859-9', 'iso-ir-148', 'iso8859-9', 'iso88599', 'iso_8859-9', 'iso_8859-9:1989', 'l5', 'latin5', 'x-cp1254' |
| 'windows-1255' | 'cp1255', 'x-cp1255' |
| 'windows-1256' | 'cp1256', 'x-cp1256' |
| 'windows-1257' | 'cp1257', 'x-cp1257' |
| 'windows-1258' | 'cp1258', 'x-cp1258' |
| 'x-mac-cyrillic' | 'x-mac-ukrainian' |
| 'gbk' | 'chinese', 'csgb2312', 'csiso58gb231280', 'gb2312', 'gb_2312', 'gb_2312-80', 'iso-ir-58', 'x-gbk' |

| Encoding | Aliases |
| --- | --- |
| 'gb18030' | |
| 'big5' | 'big5-hkscs', 'cn-big5', 'csbig5', 'x-x-big5' |
| 'euc-jp' | 'cseucpkdfmtjapanese', 'x-euc-jp' |
| 'iso-2022-jp' | 'csiso2022jp' |
| 'shift_jis' | 'csshiftjis', 'ms932', 'ms_kanji', 'shift-jis', 'sjis', 'windows-31j', 'x-sjis' |
| 'euc-kr' | 'cseuckr', 'csksc56011987', 'iso-ir-149', 'korean', 'ks_c_5601-1987', 'ks_c_5601-1989', 'ksc5601', 'ksc_5601', 'windows-949' |

*Note*: The `'iso-8859-16'` encoding listed in the WHATWG Encoding Standard is not supported.

## new TextDecoder([encoding[, options]]) #

- `encoding` `<string>` Identifies the `encoding` that this `TextDecoder` instance supports. Defaults to `'utf-8'`.
- `options` `<Object>`
  - `fatal` `<boolean>` `true` if decoding failures are fatal. Defaults to `false`. This option is only supported when ICU is enabled (see Internationalization).
  - `ignoreBOM` `<boolean>` When `true`, the `TextDecoder` will include the byte order mark in the decoded result. When `false`, the byte order mark will be removed from the output. This option is only used when `encoding` is `'utf-8'`, `'utf-16be'` or `'utf-16le'`. Defaults to `false`.

Creates an new `TextDecoder` instance. The `encoding` may specify one of the supported encodings or an alias.

## textDecoder.decode([input[, options]]) #

- `input` `<ArrayBuffer>` | `<DataView>` | `<TypedArray>` An `ArrayBuffer`, `DataView` or Typed Array instance containing the encoded data.
- `options` `<Object>`
  - `stream` `<boolean>` `true` if additional chunks of data are expected. Defaults to `false`.
- Returns: `<string>`

Decodes the `input` and returns a string. If `options.stream` is `true`, any incomplete byte sequences occuring at the end of the `input` are buffered internally and emitted after the next call to `textDecoder.decode()`.

If `textDecoder.fatal` is `true`, decoding errors that occur will result in a `TypeError` being thrown.

## textDecoder.encoding #

- `<string>`

The encoding supported by the `TextDecoder` instance.

## textDecoder.fatal #

- `<boolean>`

The value will be `true` if decoding errors result in a `TypeError` being thrown.

## textDecoder.ignoreBOM #

- `<boolean>`

The value will be `true` if the decoding result will include the byte order mark.

# Class: util.TextEncoder #

Added in: v8.3.0

An implementation of the WHATWG Encoding Standard `TextEncoder` API. All instances of `TextEncoder` only support UTF-8 encoding.

```
const encoder = new TextEncoder();
const uint8array = encoder.encode('this is some data');
```

## textEncoder.encode([input]) #

- `input` `<string>` The text to encode. Defaults to an empty string.
- Returns: `<Uint8Array>`

UTF-8 encodes the `input` string and returns a `Uint8Array` containing the encoded bytes.

## textDecoder.encoding #

- `<string>`

The encoding supported by the `TextEncoder` instance. Always set to `'utf-8'`.

# Deprecated APIs #

The following APIs have been deprecated and should no longer be used. Existing applications and modules should be updated to find alternative approaches.

## util._extend(target, source) #

Added in: v0.7.5 Deprecated since: v6.0.0

> Stability: 0 - Deprecated: Use `Object.assign()` instead.

The `util._extend()` method was never intended to be used outside of internal Node.js modules. The community found and used it anyway.

It is deprecated and should not be used in new code. JavaScript comes with very similar built-in functionality through `Object.assign()`.

## util.debug(string) #

Added in: v0.3.0 Deprecated since: v0.11.3

> Stability: 0 - Deprecated: Use `console.error()` instead.

- `string` `<string>` The message to print to `stderr`

Deprecated predecessor of `console.error`.

## util.error([...strings]) #

Added in: v0.3.0 Deprecated since: v0.11.3

> Stability: 0 - Deprecated: Use `console.error()` instead.

- `...strings` `<string>` The message to print to `stderr`

Deprecated predecessor of `console.error`.

## util.isArray(object) #

Added in: v0.6.0 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>

Internal alias for `Array.isArray`.

Returns `true` if the given `object` is an `Array`. Otherwise, returns `false`.

```
const util = require('util');

util.isArray([]);
// Returns: true
util.isArray(new Array());
// Returns: true
util.isArray({});
// Returns: false
```

## util.isBoolean(object) #

Added in: v0.11.5 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>

Returns `true` if the given `object` is a `Boolean`. Otherwise, returns `false`.

```
const util = require('util');

util.isBoolean(1);
// Returns: false
util.isBoolean(0);
// Returns: false
util.isBoolean(false);
// Returns: true
```

## util.isBuffer(object) #

Added in: v0.11.5 Deprecated since: v4.0.0

Stability: 0 - Deprecated: Use `Buffer.isBuffer()` instead.

- `object` <any>

Returns `true` if the given `object` is a `Buffer`. Otherwise, returns `false`.

```
const util = require('util');

util.isBuffer({ length: 0 });
// Returns: false
util.isBuffer([]);
// Returns: false
```

```
util.isBuffer(Buffer.from('hello world'));
// Returns: true
```

## util.isDate(object) #

Added in: v0.6.0 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>

Returns `true` if the given `object` is a `Date`. Otherwise, returns `false`.

```
const util = require('util');

util.isDate(new Date());
// Returns: true
util.isDate(Date());
// false (without 'new' returns a String)
util.isDate({});
// Returns: false
```

## util.isError(object) #

Added in: v0.6.0 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object <any>`

Returns `true` if the given `object` is an `Error`. Otherwise, returns `false`.

```
const util = require('util');

util.isError(new Error());
// Returns: true
util.isError(new TypeError());
// Returns: true
util.isError({ name: 'Error', message: 'an error occurred' });
// Returns: false
```

Note that this method relies on `Object.prototype.toString()` behavior. It is possible to obtain an incorrect result when the `object` argument manipulates `@@toStringTag`.

```
const util = require('util');
const obj = { name: 'Error', message: 'an error occurred' };

util.isError(obj);
// Returns: false
obj[Symbol.toStringTag] = 'Error';
util.isError(obj);
// Returns: true
```

## util.isFunction(object) #

Added in: v0.11.5 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>

Returns `true` if the given `object` is a `Function`. Otherwise, returns `false`.

```
const util = require('util');

function Foo() {}
const Bar = () => {};

util.isFunction({});
// Returns: false
util.isFunction(Foo);
// Returns: true
util.isFunction(Bar);
// Returns: true
```

## util.isNull(object) #

Added in: v0.11.5 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>

Returns `true` if the given `object` is strictly `null`. Otherwise, returns `false`.

```
const util = require('util');

util.isNull(0);
// Returns: false
util.isNull(undefined);
// Returns: false
util.isNull(null);
// Returns: true
```

## util.isNullOrUndefined(object) #

Added in: v0.11.5 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>

Returns `true` if the given `object` is `null` or `undefined`. Otherwise, returns `false`.

```
const util = require('util');

util.isNullOrUndefined(0);
// Returns: false
util.isNullOrUndefined(undefined);
// Returns: true
```

```
util.isNullOrUndefined(null);
// Returns: true
```

## util.isNumber(object) #

Added in: v0.11.5 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>

Returns `true` if the given `object` is a `Number`. Otherwise, returns `false`.

```
const util = require('util');

util.isNumber(false);
// Returns: false
util.isNumber(Infinity);
// Returns: true
util.isNumber(0);
// Returns: true
util.isNumber(NaN);
// Returns: true
```

## util.isObject(object) #

Added in: v0.11.5 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>

Returns `true` if the given `object` is strictly an `Object` **and** not a `Function`. Otherwise, returns `false`.

```
const util = require('util');

util.isObject(5);
// Returns: false
util.isObject(null);
// Returns: false
util.isObject({});
// Returns: true
util.isObject(function() {});
// Returns: false
```

## util.isPrimitive(object) #

Added in: v0.11.5 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>

Returns `true` if the given `object` is a primitive type. Otherwise, returns `false`.

```
const util = require('util');

util.isPrimitive(5);
// Returns: true
util.isPrimitive('foo');
// Returns: true
util.isPrimitive(false);
// Returns: true
util.isPrimitive(null);
// Returns: true
util.isPrimitive(undefined);
// Returns: true
util.isPrimitive({});
// Returns: false
util.isPrimitive(function() {});
// Returns: false
util.isPrimitive(/^$/);
// Returns: false
util.isPrimitive(new Date());
// Returns: false
```

## util.isRegExp(object) #

Added in: v0.6.0 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>

Returns `true` if the given `object` is a `RegExp`. Otherwise, returns `false`.

```
const util = require('util');

util.isRegExp(/some regexp/);
// Returns: true
util.isRegExp(new RegExp('another regexp'));
// Returns: true
util.isRegExp({});
// Returns: false
```

## util.isString(object) #

Added in: v0.11.5 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>

Returns `true` if the given `object` is a `string`. Otherwise, returns `false`.

```
const util = require('util');

util.isString('');
// Returns: true
util.isString('foo');
```

```
// Returns: true
util.isString(String('foo'));
// Returns: true
util.isString(5);
// Returns: false
```

## util.isSymbol(object) #

Added in: v0.11.5 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object` <any>

Returns `true` if the given `object` is a `Symbol`. Otherwise, returns `false`.

```
const util = require('util');

util.isSymbol(5);
// Returns: false
util.isSymbol('foo');
// Returns: false
util.isSymbol(Symbol('foo'));
// Returns: true
```

## util.isUndefined(object) #

Added in: v0.11.5 Deprecated since: v4.0.0

Stability: 0 - Deprecated

- `object <any>`

Returns `true` if the given `object` is `undefined`. Otherwise, returns `false`.

```
const util = require('util');

const foo = undefined;
util.isUndefined(5);
// Returns: false
util.isUndefined(foo);
// Returns: true
util.isUndefined(null);
// Returns: false
```

## util.log(string) #

Added in: v0.3.0 Deprecated since: v6.0.0

Stability: 0 - Deprecated: Use a third party module instead.

- `string <string>`

The `util.log()` method prints the given `string` to `stdout` with an included timestamp.

```
const util = require('util');

util.log('Timestamped message.');
```

## util.print([...strings]) #

Added in: v0.3.0 Deprecated since: v0.11.3

> Stability: 0 - Deprecated: Use `console.log()` instead.

Deprecated predecessor of `console.log`.

## util.puts([...strings]) #

Added in: v0.3.0 Deprecated since: v0.11.3

> Stability: 0 - Deprecated: Use `console.log()` instead.

Deprecated predecessor of `console.log`.